Contents

Background

What are Recommender Systems?

Recommender systems are machine-learning based algorithms that can 'learn' what a user might like to see, based on the information it has about the user or the product/item. Recommender systems can be seen in many services that we use regularly, such as Amazon recommending products to us or Facebook recommending people we may know. Some companies rely heavily on the use of these techniques and algorithms, such as Netflix, where their business model and its success are influenced by the accuracy of their recommendations.

How do Recommender Systems Work?

There are several approaches that can be taken to build a recommender system. Some of which are:

- **Simple Recommenders**: These are the easiest to implement. This is a simple recommender system that shows the user what is popular. This is a generalised method of recommending, although the idea behind this system is that things that are more popular and more liked will be more likely to be enjoyed by the average audience.
- **Collaborative Filtering**: This method is more complex than the above Simple Recommender, as it is more specific to the user. This method ignores information about the target item (song, movie, etc.), and just bases its recommendation purely on user information/opinions. The concept of collaborative filtering is based on the idea that user A has the same likes/dislikes/opinions as user B, that user A is more likely to have the same opinion as user B on a different issue than a randomly chosen user. (Interesting link about user-based vs. item-based)
- **Content-Based Recommenders**: This approach uses the item's metadata, for example the metadata held for a movie would be it's genre, director, description or actors. This algorithm identifies the genres/directors/actors that a user generally likes, and uses this as a guide for future recommendations.
- **Hybrid Approach**: This approach is a combination of collaborative filtering and content-based recommenders. This can be quite difficult to implement, although when optimized, can outperform both methods of recommending.

About the Data used in our Recommender System

Our Recommender System is based on artists from the LastFM dataset. The LastFM dataset contains information about 92,800 artists and 1,892 users. The data is comprised of 6 files:

- artists.dat : This file contains information about the artist, such as their ID, their name, and URLs associated with them
- tags.dat : This file contains a mapping for each tag ID and the tag description
- user_artists.dat : This file contains information about how many times a user has listened to each artist
- user_friends.dat : This file contains a mapping for a user and each of their friends
- user_taggedartists.dat : This file contains information about the tags assigned to each artist from each user
- user_taggedartists_timestamps.dat : This file contains the same content as user_taggedartists.dat, but with the added timestamp

The dataset also includes a readme file to provide context and further explanation on the information contained in each file.

This dataset is available through this link, or you can download it directly by clicking here.

Data Exploration

Importing Relevant Packages and Datasets

```
import pandas as pd
import numpy as np
import itertools
import matplotlib.pyplot as plt
```

```
dataframe_names = [
    'user_friends',
    'user_taggedartists',
    'artists',
    'tags',
    'user_artists']

file_names = [
    '../data/user_friends.csv',
    '../data/user_taggedartists.csv',
    '../data/artists.csv',
    '../data/tags.csv',
    '../data/user_artists.csv']
```

```
for (dataframe, file) in zip(dataframe_names, file_names):
    vars()[dataframe] = pd.read_csv(file)
```

Exploring the Users' Friends Dataset

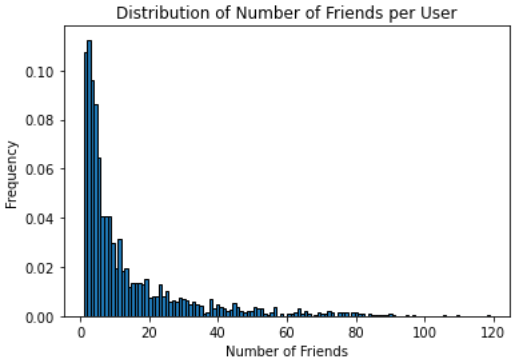
```
users_friends_count = (
    user_friends
    .groupby('userID', as_index=False)
    .agg({'friendID': ['count']})
)
```

```
users_friends_count.columns = ['userID', 'count_friends']
```

```
mean = str(users_friends_count['count_friends'].mean())
min = str(users_friends_count['count_friends'].min())
max = str(users_friends_count['count_friends'].max())

print('The average number of friends each user has is ' + mean + ' friends')
print('The user with the least amount of friends has ' + min + ' friends')
print('The user with the most amount of friends has ' + max + ' friends')
```

```
The average number of friends each user has is 13.44291754756871 friends
The user with the least amount of friends has 1 friends
The user with the most amount of friends has 119 friends
```

```
data = users_friends_count['count_friends']
plt.hist(data, weights=np.ones(len(data)) / len(data), bins=120, edgecolor='black')
plt.gca().set(
    title='Distribution of Number of Friends per User',
    xlabel='Number of Friends',
    ylabel='Frequency'
);
```



As you can see in the above chart, over 60% of the users in this dataset have between 0 and 10 friends. Only ~15% of users have between 10 and 20 friends.

We can also conclude that only less than 25% of users have over 20 friends. This is useful to know, as this would imply that a recommender system based off a user's friends' favourite artists alone may not perform accurately.

Although using friends' liked artists as a sole method to recommend songs will not perform well, it might be useful to put a slightly larger weight on their friends' artists, as long as they have a similar taste in music.

Exploring Users' Tagged Artists

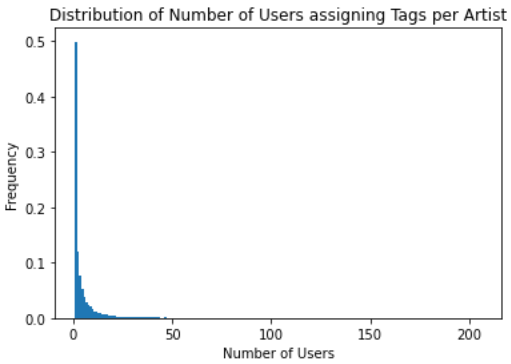
```
tagged_artists_users = (
    user_taggedartists
    .groupby('artistID', as_index=False)
    .agg({'userID': ['count']})
)

tagged_artists_users = (
    user_taggedartists
    .groupby('artistID', as_index=False)
    .agg({'userID': lambda x: x.nunique()})
)
```

```
tagged_artists_users.columns = ['artistID', 'count_users']
```

```
mean = str(tagged_artists_users['count_users'].mean())
min = str(tagged_artists_users['count_users'].min())
max = str(tagged_artists_users['count_users'].max())

print('The average artist has been assigned a tag from ' + mean + ' users')
print('The user with the least amount of of users assigning tags has ' + min + ' user(s)')
print('The user with the most amount users assigning tags has ' + max + ' users')
```

```
The average artist has been assigned a tag from 5.7730559843991225 users
The user with the least amount of of users assigning tags has 1 user(s)
The user with the most amount users assigning tags has 206 users
```

```
data = tagged_artists_users['count_users']
plt.hist(data, weights=np.ones(len(data)) / len(data), bins=200)
plt.gca().set(
    title='Distribution of Number of Users assigning Tags per Artist',
    xlabel='Number of Users',
    ylabel='Frequency'
);
```



The plot above shows that there are very few artists that have over 50 users assigning tags.

```
tagged_artists_users[tagged_artists_users['count_users']>=50].count()
```

```
artistID       146
count_users    146
dtype: int64
```

There are 55 artists who have been assigned tags from over 50 users. This suggests that there isn't a mistaken outlier, and that the data is just very sparse (just over 1%) after reaching 50 users.

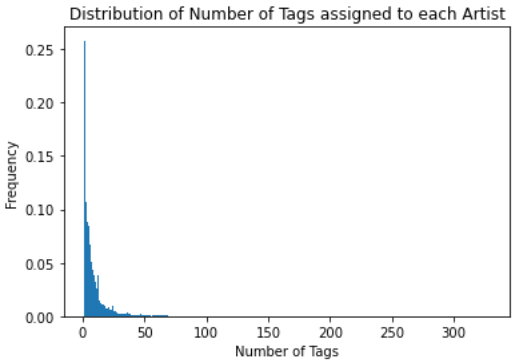
```
tagged_artists_tags = (
    user_taggedartists
    .groupby('artistID', as_index=False)
    .agg({'tagID': lambda x: x.nunique()})
)
```

```
tagged_artists_tags.columns = ['artistID', 'count_tags']
```

```
mean = str(tagged_artists_tags['count_tags'].mean())
min = str(tagged_artists_tags['count_tags'].min())
max = str(tagged_artists_tags['count_tags'].max())

print('The average artist has been assigned ' + mean + ' tags')
print('The user with the least amount of tags has ' + min + ' tag(s)')
print('The user with the most amount tags has ' + max + ' tags')
```

```
The average artist has been assigned 8.915576501178192 tags
The user with the least amount of tags has 1 tag(s)
The user with the most amount tags has 329 tags
```

```
data = tagged_artists_tags['count_tags']
plt.hist(data, weights=np.ones(len(data)) / len(data), bins=300)
plt.gca().set(
    title='Distribution of Number of Tags assigned to each Artist',
    xlabel='Number of Tags',
    ylabel='Frequency'
);
```



The above graph shows that over 25% of artists have received only 1 unique tag.

The frequency of tags seems to be very close to 0 when the number of tags is greater than 75.

```
tagged_artists_tags[tagged_artists_tags['count_tags']<=75].count()/len(tagged_artists_tags)
```

```
artistID      0.9935
count_tags    0.9935
dtype: float64
```

```
tagged_artists_tags[tagged_artists_tags['count_tags']<=5].count()/len(tagged_artists_tags)
```

```
artistID      0.537093
count_tags    0.537093
dtype: float64
```

Only 81 (0.65%) of the artists have greater than 75 unique tags assigned to them.

In fact, over half of the artists have been assigned between 1 and 5 unique tags.

These tags will allow for a more complex recommendation to be made to a user.

Exploring the Artists and Tags Dataset

```
artists.head()
```

	id	name	url	pictureURL
0	0	MALICE MIZER	http://www.last.fm/music/MALICE+MIZER	http://userserve-ak.last.fm/serve/252/10808.jpg
1	1	Diary of Dreams	http://www.last.fm/music/Diary+of+Dreams	http://userserve-ak.last.fm/serve/252/3052066.jpg
2	2	Carpathian Forest	http://www.last.fm/music/Carpathian+Forest	http://userserve-ak.last.fm/serve/252/40222717...
3	3	Moi dix Mois	http://www.last.fm/music/Moi+dix+Mois	http://userserve-ak.last.fm/serve/252/54697835...
4	4	Bella Morte	http://www.last.fm/music/Bella+Morte	http://userserve-ak.last.fm/serve/252/14789013...

```
tags.head()
```

	tagID	tagValue
0	1	metal
1	2	alternative metal
2	3	goth rock
3	4	black metal
4	5	death metal

The artists and tags datasets are just a reference table to expand on the artist details or the tag details.

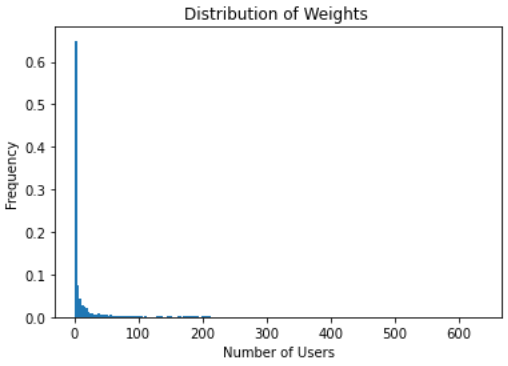
This does not require any more investigation, as there are no relevant data points to be used in our recommender system.

Exploring Users' Liked Artists

```
distribution_of_weights_given = (
    user_artists
    .groupby('weight', as_index=False)
    .agg({'userID': ['count']})
)
```

```
distribution_of_weights_given.columns = ['weight', 'users_count']
```

```
data = distribution_of_weights_given['users_count']
plt.hist(data, weights=np.ones(len(data)) / len(data), bins=200)
plt.gca().set(
    title='Distribution of Weights ',
    xlabel='Number of Users',
    ylabel='Frequency'
);
```

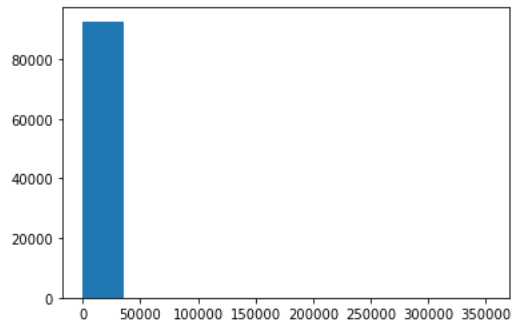



```
user_artists['weight']
```

```
0        13883
1        11690
2        11351
3        10300
4         8983
         ...
92829      337
92830      297
92831      281
92832      280
92833      263
Name: weight, Length: 92834, dtype: int64
```

```
list_of_weights = np.asarray(user_artists['weight'])
```

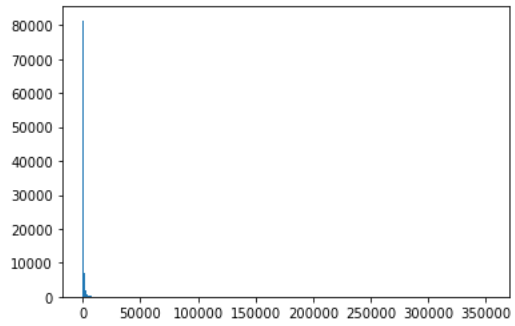
```
plt.hist(list_of_weights);
```




```
list_of_weights.mean()
```

```
745.2439300256372
```

```
plt.hist(list_of_weights, bins=300);
```



Let's find out what proportion of people's weights are under 10,000

```
user_artists[user_artists['weight']<=10000].count()/92834
```

```
userID      0.993214
artistID    0.993214
weight      0.993214
dtype: float64
```

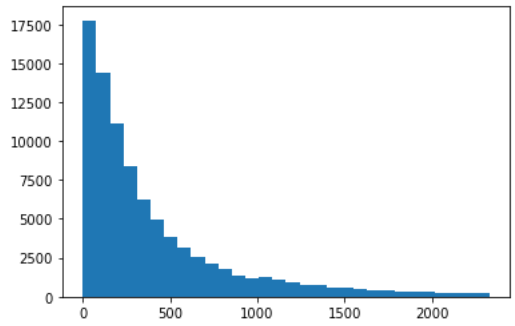
99.32% of people are under 10,000 listens

```
user_artists[user_artists['weight']<=2328].count()/92834
```

```
userID      0.950008
artistID    0.950008
weight      0.950008
dtype: float64
```

95% of the users listen to artists under 2328 times

```
plt.hist(user_artists[user_artists['weight']<=2328]['weight'], bins = 30);
```



Adjusting User-Artist Weights

Importing Relevant Packages and Datasets

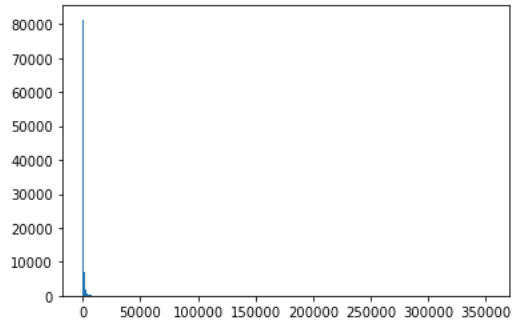
```
import pandas as pd
import numpy as np
import itertools
import matplotlib.pyplot as plt
from functions import *
```

```
dataframe_names = [
    'user_friends',
    'user_taggedartists',
    'artists',
    'tags',
    'user_artists']

file_names = [
    '../data/user_friends.csv',
    '../data/user_taggedartists.csv',
    '../data/artists.csv',
    '../data/tags.csv',
    '../data/user_artists.csv']
```

```
for (dataframe, file) in zip(dataframe_names, file_names):
    vars()[dataframe] = pd.read_csv(file)
```

Exploring the data

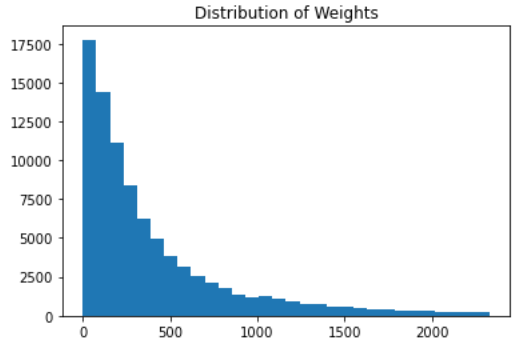
```
plt.hist(user_artists['weight'], bins=300);
```



It is difficult to visualise the distribution of the weights using this graph. It is likely that there are few datapoints that go beyond 35,000 and this is causing the rest of the data to be very small and difficult to interpret.

By running some basic commands, I have discovered that 95% of weights are less than or equal to 2,328.

```
plt.hist(user_artists[user_artists['weight']<=2328]['weight'], bins = 30)
plt.gca().set(title='Distribution of Weights');
```



This is the new plot where we have excluded any weights over 2,328.

The graph is much easier to interpret and seems to follow a gamma/exponential distribution.

Different options

At the moment, it would be quite difficult to seperate the data as it does not follow a normal distribution.

There are different methods we can try to assign a rating for each weight.

Option 1

Since the distribution of weights has a very long tail, we can find a point where most of the tail is where users gave artists a 5-star rating, and then divide the rest of the weights intuitively/mathematically into 1-4 stars. This would be to assume a 25% split between 1-star, 2-star, 3-star and 4-star ratings (after we have identified the cut-off poiint for 5-star ratings).

The drawbacks of this option:

- It is not user specific, the percentages (25%) will be the same for all users, except the 5-star rating, which may be disproportionately distributed if someone listens to a huge amount of music.
- It is not accurately representative of natural distribution you'd expect from users liking artists. It is more likely to follow a slight normal distribution where there are more 3 star ratings than 1 or 5 star ratings.

Option 2

Use a user's average weighting and do a normal distribution seperator.

1. Do a case study on 10 random people to see the distribution of their weights
2. If the distribution is normal, follow the standard procedure to separate the ratings, if the distribution is not normal, try to identify if the distribution could be separated in a different way.

The drawbacks of this option:

- It is unlikely that the users' specific weight distribution is normal as the weight distribution as a whole is extremly long-tailed.
- If we are working with a long-tailed distribution, the method to assign ratings will have to be manually decided and might not be optimal.

Option 2 seems like it will yield better results.

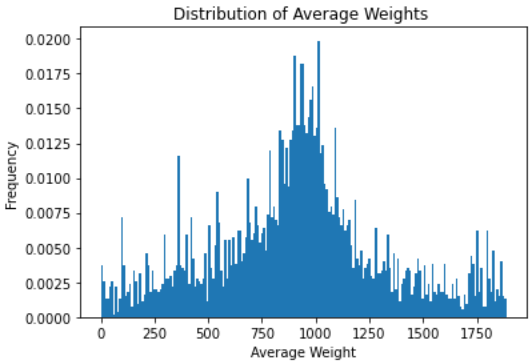
We'll firstly investigate the distribution of all users' average weights and then look at 5 users and view their weight distributions in isolation.

Investigating Users' Average Weight

```
distribution_of_weights_given = (
    user_artists
    .groupby('weight', as_index=False)
    .agg({'userID': ['mean']})
)
```

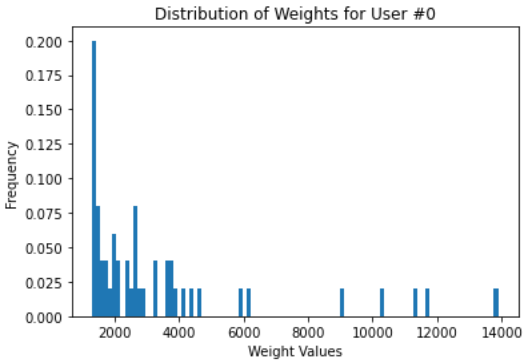
```
distribution_of_weights_given.columns = ['weight', 'users_avg']
```

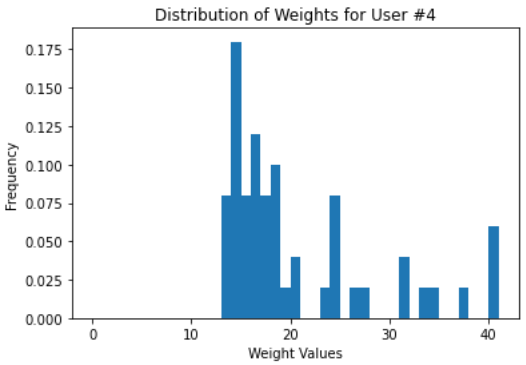
```
data = distribution_of_weights_given['users_avg']
plt.hist(data, weights=np.ones(len(data)) / len(data), bins=200)
plt.gca().set(
    title='Distribution of Average Weights ',
    xlabel='Average Weight',
    ylabel='Frequency'
);
```

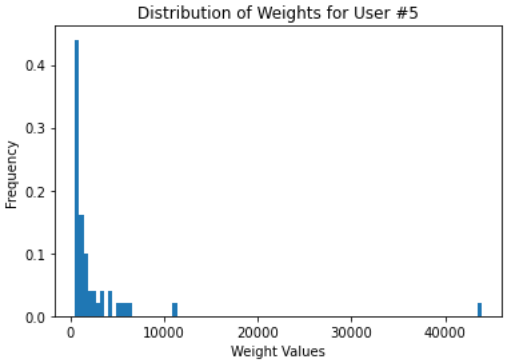


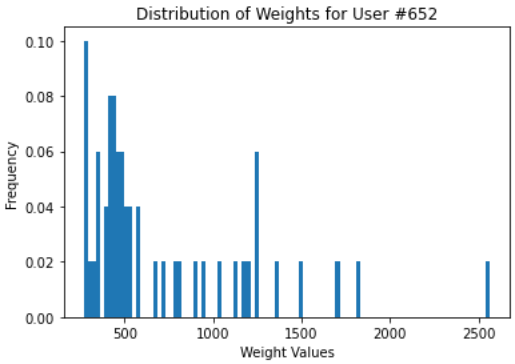
This graph shows quite a clear normal distribution, this may be useful to us depending on the specific user distributions.

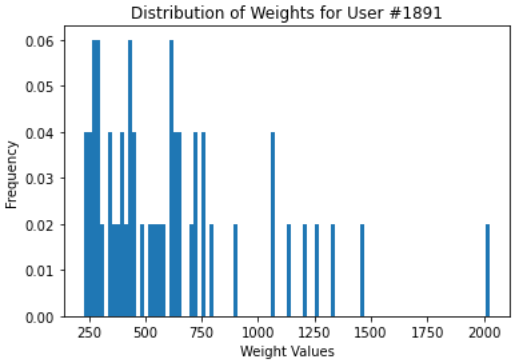
Investigating 5 Users' Distributions









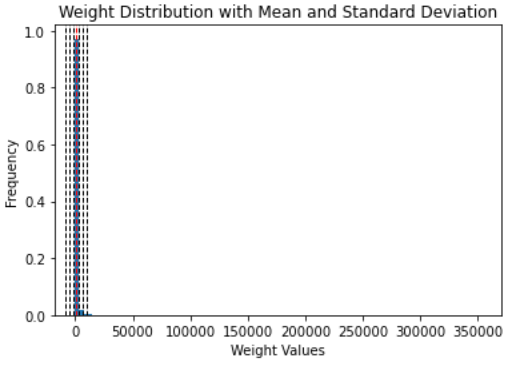


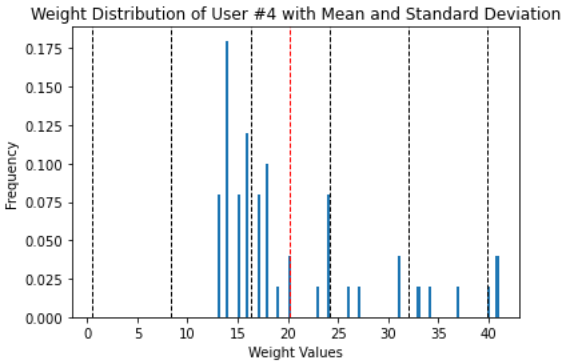
All of the above figures, for users #0, #4, #5, #652, #1891, have a long-tailed distribution.

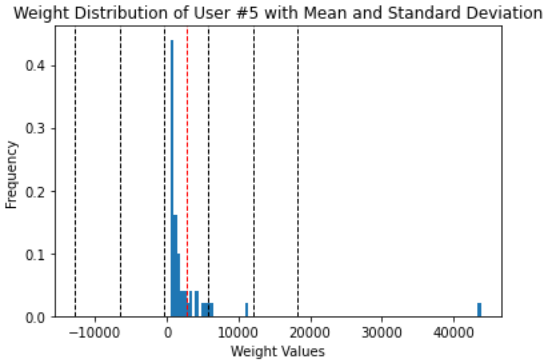
This makes the distribution of ratings slightly more tricky.

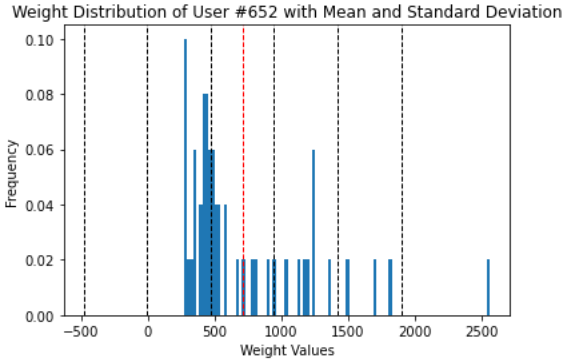
Let's investigate for some of these users, how they would be processed if treated like a normal distribution.

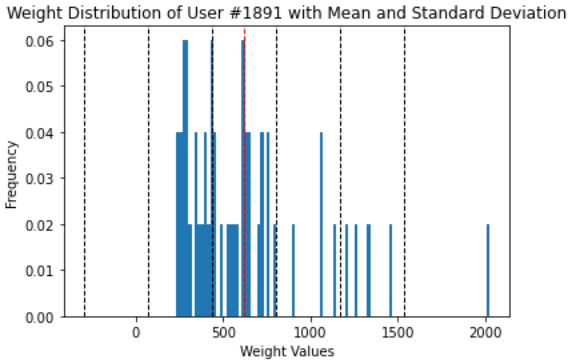
Plotting the Mean and Standard Deviation on Users' Weight Distribution











These five graphs show us that using the mean and standard deviation as a method to assign our-star ratings, it would have some key performance issues.

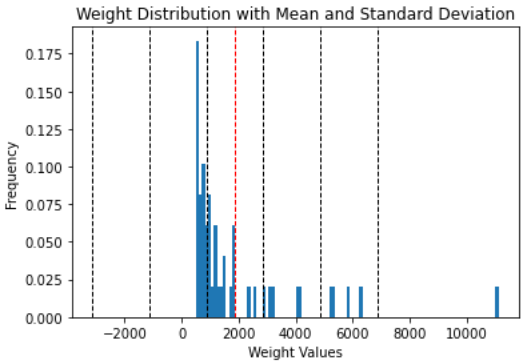
This method does not prove very efficient as there are no occurrences of a 1-star rating for all users examined. It also does not apply to users who have outlier-like values, such as user #5 having one occurence of a weight above 40,000 which increases the standard deviation substantially, and results in majority of their weights belonging to the 3-star rating zone.

This issue with user #5 could be solved with outlier exclusion. Excluding outliers will reduce the size of the standard deviation, allowing the assignment of ratings to be more spread out, which may solve the issue of 1-star ratings also.

Removing Outliers for User #5

```
user5 = remove_users_outliers(user_artists, 5)
user5_mean = user5['weight'].mean()
user5_std = np.std(user5['weight'])
```

```
plot_weights_mean_and_std(user5)
```



The above graph shows slight improvement in comparison to the original graph contained in this section where the means and standard deviations were graphed.

Using the adjusted data, the datapoint above 1,000 could be considered a datapoint, and we can iterate through the outlier exclusion process again. We can explore this later.

Since there is only a sample set of 5 users, they may not be representative of the group of users. Let's assign the user ratings according to the normal distribution (mean and standard deviation), and see what adjustments can be made post-hoc.

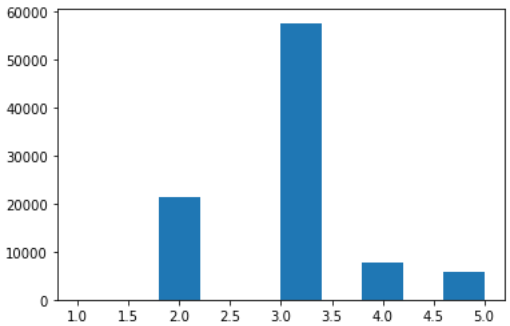
Creating a Loop to add Star-Ratings based on Normal Distribution

```
means_and_stds = find_means_and_stds_per_user(user_artists)
means = means_and_stds[0]
stds = means_and_stds[1]
```

```
user_artists['rating'] = 0
user_artists['mean'] = 0
user_artists['std'] = 0
```

```
user_artists['mean'] = user_artists.apply(lambda row: find_mean(row, means), axis=1)
user_artists['std'] = user_artists.apply(lambda row: find_std(row, stds), axis=1)
user_artists['rating'] = user_artists.apply(lambda row: find_rating(row), axis=1)
```

```
plt.hist(user_artists['rating']);
```



There are no artists that recieved a 1-star rating, this is due to the fact that the users' weights are not normally distributed, they're long-tailed distributions.

This causes the standard deviation to be larger, meaning that most of the weights to the left of the mean will be within 1 standard deviation of the mean, resulting in no 1-star ratings.

We can check now if removing the outliers will make a difference to the distribution of star ratings.

Adjusting Star-Ratings to Allow for Outlier Exclusion

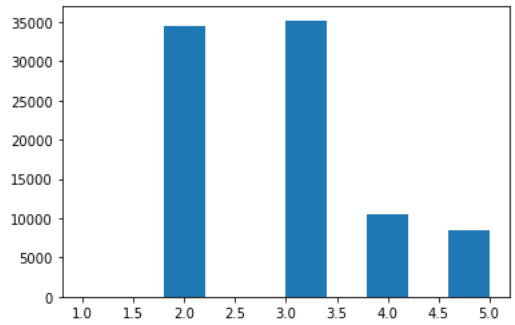
```
## Removing any weight that is above 2 standard deviations away from the mean
adjusted_user_artists = user_artists[abs(user_artists['weight']-user_artists['mean'])
<(2*user_artists['std'])]
```

```
## Dropping and recreating the mean, standard deviation and rating columns
adjusted_user_artists = adjusted_user_artists.drop(['std', 'rating', 'mean'], axis=1)

adjusted_user_artists['rating'] = 0
adjusted_user_artists['mean'] = 0
adjusted_user_artists['std'] = 0
```

```
means_and_stds = find_means_and_stds_per_user(adjusted_user_artists)
means = means_and_stds[0]
stds = means_and_stds[1]
```

```
## Recalculate the means, standard deviations and ratings

adjusted_user_artists['mean'] = adjusted_user_artists.apply(lambda row: find_mean(row, means),
axis=1)
adjusted_user_artists['std'] = adjusted_user_artists.apply(lambda row: find_std(row, stds),
axis=1)
adjusted_user_artists['rating'] = adjusted_user_artists.apply(lambda row: find_rating(row),
axis=1)
```

```
plt.hist(adjusted_user_artists['rating'], bins=10);
```



The adjustment didn't have a huge effect on the final allocation of ratings, there are still a large number of weights on the tail of the distribution that are skewing the ratings.

A next possible option would be to filter the data where weights are below 2328, as we found that this exludes the top 5% of values that could be skewing the data.

Adding Star Ratings Excluding the Top 5% of Weights

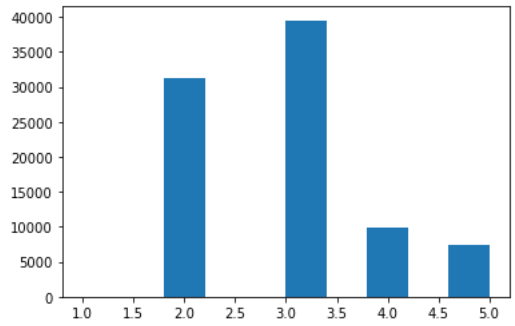
```
adjusted_user_artists = user_artists[user_artists['weight']<=2328]
```

```
## Dropping and recreating the mean, standard deviation and rating columns
adjusted_user_artists = adjusted_user_artists.drop(['std', 'rating', 'mean'], axis=1)

adjusted_user_artists['rating'] = 0
adjusted_user_artists['mean'] = 0
adjusted_user_artists['std'] = 0
```

```
## Finding the new means and standard deviations
means_and_stds = find_means_and_stds_per_user(adjusted_user_artists)
means = means_and_stds[0]
stds = means_and_stds[1]
```

```
## Recalculate the means, standard deviations and ratings
adjusted_user_artists['mean'] = adjusted_user_artists.apply(lambda row: find_mean(row, means),
axis=1)
adjusted_user_artists['std'] = adjusted_user_artists.apply(lambda row: find_std(row, stds),
axis=1)
adjusted_user_artists['rating'] = adjusted_user_artists.apply(lambda row: find_rating(row),
axis=1)
```

```
plt.hist(adjusted_user_artists['rating']);
```



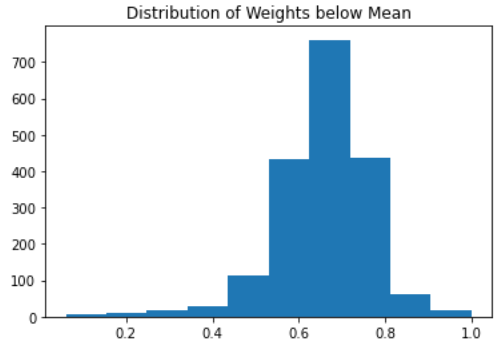
Still have the same issue, let's investigate what proportion of users' weights are less than their mean.

Investigating Proportion of Weights below Mean

```
## Finding the new means and standard deviations
means_and_stds = find_means_and_stds_per_user(adjusted_user_artists)
means = means_and_stds[0]
stds = means_and_stds[1]
```

```
below_mean = {}
weights = []
for userID in adjusted_user_artists.userID.unique():
    total = 0
    below_means_count = 0
    user_data = user_artists[user_artists['userID']==userID]
    for weight in user_data['weight']:
        if weight <= means[userID]:
            below_means_count += 1
            weights.append(weight)
        total += 1
    below_mean[userID] = below_means_count/total
```

```
plt.hist(below_mean.values())
plt.gca().set(title='Distribution of Weights below Mean');
```



Judging by this graph, almost all of the users have most of their weights lower than their mean, meaning that there are few very large weights that are causing this increase in the mean.

Final Decision

There is no one-size-fits-all for each user, as each user's listening habits are different.

The standard deviation cannot be used as the values to the left of the mean are distributed differently to the values on the right side of the mean, so we should introduce a split at the mean.

1. Find the mean
2. Find the minimum, and find the range of weights between the minimum and the mean.
3. Adjust the ratings so this range is divided into 25%, 50%, 25% for 1-star, 2-star and 3-star ratings respectively.
4. Find the maximum (up to a limit), and find the range of weights between the mean and the maximum.
5. Adjust the ratings so this range is divided into 25%, 50%, 25% for 3-star, 4-star and 5-star ratings respectively.

The weights will not be distributed evenly, the percentages will be calculated as minimum + ((mean - min) * 0.25) for the 1-star ratings, for example.

```
user_artists.drop('std', axis=1, inplace=True)
user_artists['mean'] = 0
user_artists['min'] = 0
user_artists['max'] = 0
user_artists['rating'] = 0
```

```
## Creating a dictionary to discover each users mean and standard deviation
means = {}
mins = {}
maxs = {}
for userID in user_artists.userID.unique():
    user_data = user_artists[user_artists['userID']==userID]
    user_mean = user_data['weight'].mean()
    user_min = user_data['weight'].min()
    user_max = user_data['weight'].max()
    means[userID] = user_mean
    mins[userID] = user_min
    maxs[userID] = user_max
```

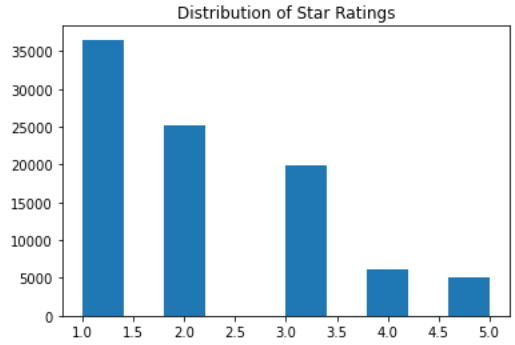
```
def find_mean(row, dict):
    return dict[row['userID']]

def find_min(row, dict):
    return dict[row['userID']]

def find_max(row, dict):
    return dict[row['userID']]
```

```
user_artists['mean'] = user_artists.apply(lambda row: find_mean(row, means), axis=1)
user_artists['min'] = user_artists.apply(lambda row: find_min(row, mins), axis=1)
user_artists['max'] = user_artists.apply(lambda row: find_max(row, maxs), axis=1)
```

```
def find_rating(row):
    weight = row['weight']
    user_mean = row['mean']
    user_min = row['min']
    user_max = row['max']
    ## Adjusting the max to 3000 to preserve distribution
    if user_max > 3000:
        user_max = 3000
    if weight <= (user_min + ((user_mean - user_min) * 0.25)):
        return 1
    elif weight <= (user_min + ((user_mean - user_min) * 0.75)):
        return 2
    elif weight <= user_max - ((user_max - user_mean) * 0.75):
        return 3
    elif weight <= user_max - ((user_max - user_mean) * 0.25):
        return 4
    else:
        return 5
```

```
user_artists['rating'] = user_artists.apply(lambda row: find_rating(row), axis=1)
```

```
plt.hist(user_artists['rating'])
plt.gca().set(title='Distribution of Star Ratings');
```



This method seems to have an accurate representation of ratings, it is easier to listen to several artists just once and skip them than it is to listen to several artists thousands of times each.

We need to use these ratings for our recommender system, so we will remove unnecessary columns from the dataframe and export it as a CSV file into our data directory.

```
user_artists.drop(['mean', 'min', 'max'], axis = 1, inplace = True)
```

```
user_artists
```

	userID	artistID	weight	rating
0	0	45	13883	5
1	0	46	11690	5
2	0	47	11351	5
3	0	48	10300	5
4	0	49	8983	5
...	...	...	...	...
92829	1891	17615	337	2
92830	1891	17616	297	1
92831	1891	17617	281	1
92832	1891	17618	280	1
92833	1891	17619	263	1

92834 rows × 4 columns

```
user_artists.to_csv('../data/user_artists_ratings.csv', index=False)
```

Assigning Tags to Artists

Importing Relevant Packages and Datasets

```
import pandas as pd
import numpy as np
import heapq
```

```
artists = pd.read_csv('../data/artists.csv')
user_taggedartists = pd.read_csv('../data/user_taggedartists.csv')
tags = pd.read_csv('../data/tags.csv')
user_artists_ratings = pd.read_csv('../data/user_artists_ratings.csv')
```

Finding the top 2 tags per artist

```
artists['first_tag'] = 0
artists['second_tag'] = 0
```

```
for artist in user_taggedartists.artistID.unique():
    artist_data = user_taggedartists[user_taggedartists['artistID']==artist]
    if len(artist_data) < 1:
        first_tag = 0
        second_tag = 0
    else:
        top_two_tags = artist_data['tagID'].value_counts()[:3].index.tolist()
        first_tag = top_two_tags[0]
        if len(top_two_tags)>1:
            second_tag = top_two_tags[1]
        else:
            second_tag = 0
    artists.loc[artists.id == artist, 'first_tag'] = first_tag
    artists.loc[artists.id == artist, 'second_tag'] = second_tag
```

Adding a Column for the Top 5 Tags (String Values)

```
tag_dict = {}
tag_dict[0] = ""
for row in tags.itertuples():
    tagID = row[1]
    tag = row[2]
    tag_dict[tagID] = tag
```

```
artists['top_five_tags'] = ''
```

```
for artist in user_taggedartists.artistID.unique():
    artist_data = user_taggedartists[user_taggedartists['artistID']==artist]
    top_five_tags = artist_data['tagID'].value_counts()[:5].index.tolist()

    string_tags = [tag_dict[tag] for tag in top_five_tags]

    top_5_string = " ".join(string_tags)
    artists.loc[artists.id == artist, 'top_five_tags'] = top_5_string
```

Exporting the CSV File

```
artists.to_csv('../data/artists_tags.csv', index=False)
```

Basic Recommender System

Importing Relevant Packages and Datasets

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
dataframe_names = [
    'user_friends',
    'user_taggedartists',
    'artists',
    'tags',
    'user_artists',
    'user_artists_ratings']

file_names = [
    '../data/user_friends.csv',
    '../data/user_taggedartists.csv',
    '../data/artists.csv',
    '../data/tags.csv',
    '../data/user_artists.csv',
    '../data/user_artists_ratings.csv']
```

```
for (dataframe, file) in zip(dataframe_names, file_names):
    vars()[dataframe] = pd.read_csv(file)
```

Recommender System Outline

For the basic recommender system, this will not take in any personalisation - it will be the same recommendations for all users.

This recommender system only recommends artists that have high ratings.

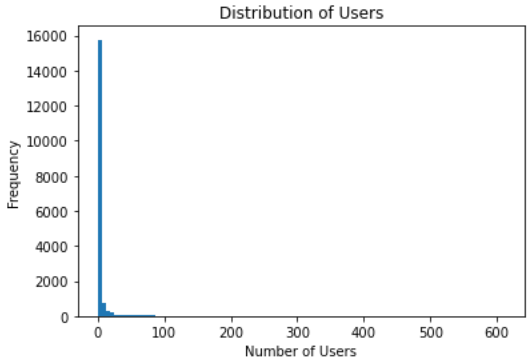
We will use the artists' average ratings, but this could be skewed by an artist who has very few ratings that yield a high average. To prevent this, we will take the top 10% of artists that have the highest number of user ratings, and then filter to find the artists with the highest average rating.

1. Make one for most popular all time
2. Make one for trending and use the timestamps one - maybe per genre?

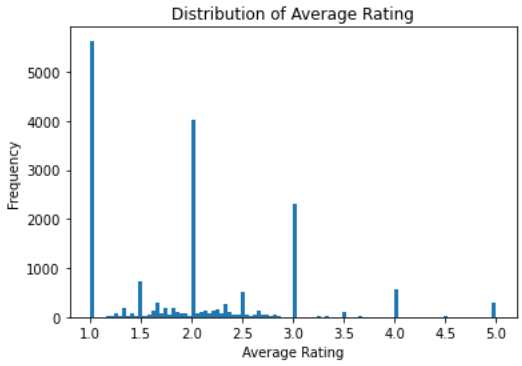
```
artists_users_ratings = (
    user_artists_ratings
    .groupby('artistID', as_index=False)
    .agg({'userID': ['count'], 'rating': ['mean']})
)
```

```
artists_users_ratings.columns = ['artistID', 'count_users', 'avg_rating']
```

```
plt.hist(artists_users_ratings['count_users'], bins=100)
plt.gca().set(
    title='Distribution of Users',
    xlabel='Number of Users',
    ylabel='Frequency'
);
```




```
plt.hist(artists_users_ratings['avg_rating'], bins=100)
plt.gca().set(
    title='Distribution of Average Rating',
    xlabel='Average Rating',
    ylabel='Frequency'
);
```



It's clear from the first graph that almost all of the artists in the dataset have less than 100 users listening to them. We can also see that the average rating received by artists has a negative linear relationship. This will make sense as users are shown lots of artists but may only listen to a small amount more regularly.

Recommending the Top 5 Artists that You Might Like

```
## Finding the top 10% of artists
np.percentile(artists_users_ratings['count_users'], 90)
```

```
8.0
```

```
len(artists_users_ratings[artists_users_ratings['count_users']==1]
['count_users'])/len(artists_users_ratings['count_users'])
```

```
0.605660163339383
```

Over 60% of artists have only had one user listen to their music.

```
most_listened_to_artists = artists_users_ratings[artists_users_ratings['count_users']>=8]
```

```
most_listened_to_artists['avg_rating'].max()
```

```
3.5
```

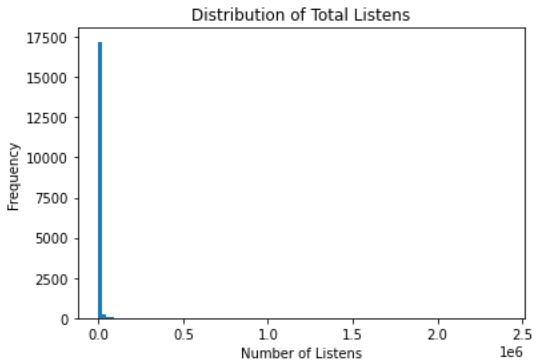
Since the highest average rating for the top 10% of artists is only 3.5, we can either increase the amount of artists being considered (increase to the top 20% of artists), or we can identify the amount of times the artists is listened to per user.

Identifying Total Listens per User

```
artists_total_listens = (
    user_artists_ratings
    .groupby('artistID', as_index=False)
    .agg({'userID': ['count'], 'rating': ['mean'], 'weight': ['sum']})
)
```

```
artists_total_listens.columns = ['artistID', 'count_users', 'avg_rating', 'total_listens']
```

```
plt.hist(artists_total_listens['total_listens'], bins=100)
plt.gca().set(
    title='Distribution of Total Listens',
    xlabel='Number of Listens',
    ylabel='Frequency'
);
```

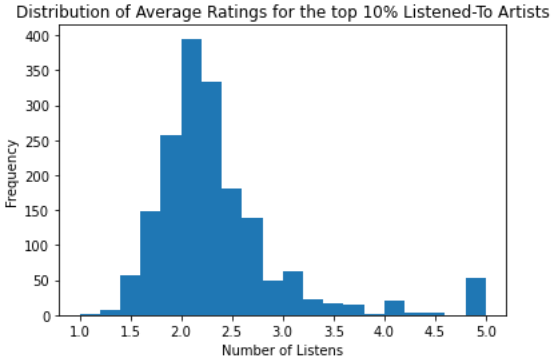



```
## Finding the top 10% of artists
np.percentile(artists_total_listens['total_listens'], 90)
```

```
4645.399999999998
```

```
artists_highest_total_listens =
artists_total_listens[artists_total_listens['total_listens']>=4645]
```

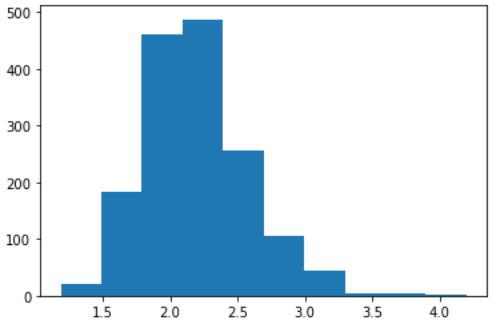
```
plt.hist(artists_highest_total_listens['avg_rating'], bins=20)

plt.gca().set(
    title='Distribution of Average Ratings for the top 10% Listened-To Artists',
    xlabel='Number of Listens',
    ylabel='Frequency'
);
```




```
artists_highest_listens =
artists_highest_total_listens[artists_highest_total_listens['count_users']>=5]
```

By filtering this dataset to only artists that have a user count above 5, removed artists that have one/two user(s) who have listened a very large number of times and have given 5-star ratings.

```
plt.hist(artists_highest_listens['avg_rating']);
```




```
recommendations = artists_highest_listens['avg_rating'].nlargest(2)
```

```
recommendations
```

```
10886    4.200000
2285     3.833333
Name: avg_rating, dtype: float64
```

```
artists_ordered_popularity = artists_highest_listens.sort_values(by='avg_rating',
ascending=False)
```

```
artist1_rec = artists_ordered_popularity.iloc[0,]['artistID']
artist2_rec = artists_ordered_popularity.iloc[1,]['artistID']
```

```
artist1_details = artists[artists['id']==artist1_rec]
artist1_name = artist1_details.values[0][1]
artist1_link = artist1_details.values[0][2]
```

```
artist2_details = artists[artists['id']==artist2_rec]
artist2_name = artist2_details.values[0][1]
artist2_link = artist2_details.values[0][2]
```

```
artist2_link
```

```
'http://www.last.fm/music/Ezginin+G%C3%BCnl%C3%BC%C4%9F%C3%BC'
```

```
print("We think you might like " + artist1_name + ", and you can view their profile through this link: " + artist1_link)
```

```
We think you might like Bernard Butler, and you can view their profile through this link: http://www.last.fm/music/Bernard+Butler
```

```
print("You might also like " + artist2_name + ", and you can view their profile through this link: " + artist2_link)
```

```
You might also like Ezginin Günlüğü, and you can view their profile through this link: http://www.last.fm/music/Ezginin+G%C3%BCnl%C3%BC%C4%9F%C3%BC
```

It might also be interesting to allow the user to decide what percentage of artists they want to see

Let's see the results for if we wanted 50% of the artists included.

```
## Finding the top 10% of artists
np.percentile(artists_users_ratings['count_users'], 10)
```

```
1.0
```

Content-Based Recommender System

Importing Relevant Packages and Datasets

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import linear_kernel
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity
```

```
dataframe_names = [
    'user_friends',
    'user_taggedartists',
    'artists',
    'tags',
    'user_artists',
    'user_artists_ratings',
    'artists_tags']

file_names = [
    '../data/user_friends.csv',
    '../data/user_taggedartists.csv',
    '../data/artists.csv',
    '../data/tags.csv',
    '../data/user_artists.csv',
    '../data/user_artists_ratings.csv',
    '../data/artists_tags.csv']
```

```
for (dataframe, file) in zip(dataframe_names, file_names):
    vars()[dataframe] = pd.read_csv(file)
```

Adjusting Existing Columns

Importing artists that have their top two tags and converting them to their string equivalents

```
tag_dict = {}
tag_dict[0] = ""
for row in tags.itertuples():
    tagID = row[1]
    tag = row[2]
    tag_dict[tagID] = tag
```

```
artists_tags['first_tag'] = artists_tags.apply(lambda row: tag_dict[row['first_tag']], axis=1)
artists_tags['second_tag'] = artists_tags.apply(lambda row: tag_dict[row['second_tag']], axis=1)
```

Recommender Based on Top Tag

We aim to calculate the Term Frequency-Inverse Document Frequency (TF-IDF) for each artist. The terms will be taken from the assigned tags for each artist.

```
tfidf = TfidfVectorizer(stop_words='english')
tfidf_matrix = tfidf.fit_transform(artists_tags['first_tag'])
tfidf_matrix.shape
```

```
(17632, 1246)
```

This 'tfidf' object was created to remove any stop words that may appear in the user-assigned tags for each artist.

This was then fitted as a matrix with the dimensions (N x M) where N is the number of artists and M is the number of tokens/words (excluding stop words) that appear in their top tag. In this case, there are 17,632 artists and 1,246 words.

```
tfidf.get_feature_names()[900:910]
```

```
['play',
 'pleasure',
 'pleasuredome',
 'pleasures',
 'podcast',
 'poetry',
 'polecane',
 'polish',
 'political',
 'pop']
```

This is an example of some of the tokens that appear in the top five tags of each artist.

There are some tags that wouldn't be classified as genres, such as 'pleasures' or 'play', although these are unlikely to affect the performance of the recommender system.

```
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_30472/2808778117.py in <module>
----> 1 cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\sklearn\metrics\pairwise.py
in linear_kernel(X, Y, dense_output)
   1003     """
   1004     X, Y = check_pairwise_arrays(X, Y)
-> 1005     return safe_sparse_dot(X, Y.T, dense_output=dense_output)
   1006
   1007

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\sklearn\utils\validation.py
in inner_f(*args, **kwargs)
     61             extra_args = len(args) - len(all_args)
     62             if extra_args <= 0:
---> 63                 return f(*args, **kwargs)
     64
     65             # extra_args > 0

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\sklearn\utils\extmath.py in
safe_sparse_dot(a, b, dense_output)
    154     if (sparse.issparse(a) and sparse.issparse(b)
    155             and dense_output and hasattr(ret, "toarray")):
--> 156         return ret.toarray()
    157     return ret
    158

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\scipy\sparse\compressed.py
in toarray(self, order, out)
   1029         if out is None and order is None:
   1030             order = self._swap('cf')[0]
-> 1031         out = self._process_toarray_args(order, out)
   1032         if not (out.flags.c_contiguous or out.flags.f_contiguous):
   1033             raise ValueError('Output array must be C or F contiguous')

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\scipy\sparse\base.py in
_process_toarray_args(self, order, out)
   1200             return out
   1201         else:
-> 1202             return np.zeros(self.shape, dtype=self.dtype, order=order)
   1203
   1204

MemoryError: Unable to allocate 2.32 GiB for an array with shape (17632, 17632) and data type
float64
```

As we are using the TF-IDF Vectoriser, we can calculate the dot product between each vector to give us the cosine similarity score. This is more efficient than to use cosine_similarities().

```
cosine_sim.shape
```

```
(17632, 17632)
```

This makes sense as this similarity is comparing each artist to each other, so each artist has it's own column and it's own row.

```
cosine_sim[2]
```

```
array([0., 0., 1., ..., 0., 0., 0.])
```

This is an example of the cosine similarities for the artist with the ID 2.

The artist will have 100% similarity with itself, hence there is 1.0 in the 2nd column.

```
indices = pd.Series(artists_tags.index, index=artists_tags['name']).drop_duplicates()
```

The above 'indices' series is created to reverse the mapping of artist names and IDs, this allows for easier searching in the recommender system.

```
def get_recommendations(name, cosine_sim=cosine_sim):
    # Get the index of the artist that matches the name
    index = indices[name]

    # Get the pairwsie similarity scores of all artists with that artist
    sim_scores = list(enumerate(cosine_sim[index]))

    # Sort the artists based on the similarity scores
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)

    # Get the scores of the 10 most similar artists
    sim_scores = sim_scores[1:11]

    # Get the artist indices
    artist_indices = [i[0] for i in sim_scores]

    # Return the top 10 most similar movies
    return artists_tags['name'].iloc[artist_indices]
```

The above function retrieves the top 10 recommendations based on the most common tag that each artist received.

This will work quite well for artists that have received lots of the same tag, making it more accurate, and may not perform as well on artists with few/no tags.

```
get_recommendations('Kanye West')
```

```
238     Flagelo Urbano
243             Common
249               Nach
253         9th Wonder
261                Nas
265            Mos Def
272               2Pac
300    Black Eyed Peas
325         Kanye West
454           Noize MC
Name: name, dtype: object
```

```
artists_tags[artists_tags['name']=='Flagelo Urbano'] ## hip hop under, raptuga, underground hip
hp, hip hop
artists_tags[artists_tags['name']=='Common'] ## hip-hop, hip hop, rap, mean, 90s
artists_tags[artists_tags['name']=='Nach'] ## hip-hop
```

	id	name	url	pictureURL	first_tag	seco
249	249	Nach	http://www.last.fm/music/Nach	http://userserve-ak.last.fm/serve/252/2510461.jpg	hip-hop	

```
get_recommendations('Coptic Rain')
```

```
1       Diary of Dreams
2     Carpathian Forest
3          Moi dix Mois
4           Bella Morte
5             Moonspell
6        Marilyn Manson
7           DIR EN GREY
8           Combichrist
9               Grendel
10            Agonoize
Name: name, dtype: object
```

```
artists_tags[artists_tags['name']=='Diary of Dreams'] ## darkwave, german, gothic, seen live,
industrial
artists_tags[artists_tags['name']=='Carpathian Forest'] ## black metal, true norwegian black
metal, norwegian
artists_tags[artists_tags['name']=='Moi dix Mois'] ## j-rock, japanese, visual kei, gothic metal
```

	id	name	url	pictureURL	first_tag	se
4	4	Bella Morte	http://www.last.fm/music/Bella+Morte	http://userserve-ak.last.fm/serve/252/14789013...	darkwave	

As you can see when we run 'Kanye West' into the recommender system, we are given lots of artist that are related to hip-hop or rap. Although, the third recommendation, Nach, only has one tag assigned to it, which suggests that Nach received very few tags, and may not be similar to Kanye West.

With our second example, we ran 'Coptic Rain' into the recommender system, to see how it would perform on an artist that had no tags. The tags of the top three recommended artists seem to be quite random (darkwave, german, norwegian, japanese). The system performed quite poorly for this artist.

Using the Top-5 Instead of Top-1 Tags

```
artists_tags['top_five_tags'] = artists_tags['top_five_tags'].fillna('')
```

```
def find_name_and_tags(dataframe):
    return "".join(dataframe['name'].split()) + ' ' + dataframe['top_five_tags']

artists_tags['name_and_tags'] = artists_tags.apply(find_name_and_tags, axis=1)
```

The name of the artist may be important to use in the recommender system as it may be referenced within some of the tags.

Some of the tags produced by the users may contain comments such as "sounds like michael jackson", and in this case it can create relations between Michael Jackson and these comments. It can also build relationships where the artist name may appear as "Jay-Z ft. Kanye West", so that Jay-Z and Kanye West could achieve a larger cosine similarity.

We are merging the name and the tags into one column to simplify things.

```
count = CountVectorizer(stop_words='english')
count_matrix = count.fit_transform(artists_tags['name_and_tags'])
```

In this case, we use CountVectorizer() instead of TF-IDF, as previously done. This is just to avoid down-weighting important tags that may appear more often than others.

```
count_matrix.shape
```

```
(17632, 20107)
```

There are much more words being produced when we accept the top 5 tags and the name of the artists.

```
cosine_sim2 = cosine_similarity(count_matrix, count_matrix)
```

```
indices = pd.Series(artists_tags.index, index=artists_tags['name'])
```

Here, we are creating the new cosine similarity matrix and the new indices object that can be passed into our original get_recommendations() function.

```
get_recommendations('Kanye West', cosine_sim2)
```

```
2578     will.i.am
16355        Trina
898      Timbaland
1435          Akon
3280         Drake
6425         Diddy
7596        Chingy
7968        T-Pain
7991        Fugees
11483         Iyaz
Name: name, dtype: object
```

```
get_recommendations('Coptic Rain', cosine_sim2)
```

```
15869        City Rain
7461              Rain
9187     The Mavericks
12739      Room Eleven
1469       B.J. Thomas
12457       Black Rain
954              Yanni
2053            Yiruma
3537        Gary Allan
3559       Clint Black
Name: name, dtype: object
```

When we run 'Kanye West' in the new recommender system, it seems to output more accurate results, the top 10 similar artists seem to be more similar this time.

However, when we run 'Coptic Rain', who has no tags, we get other artists with very similar names, as it is the only token being considered.

Collaborative Filtering Recommender System

Importing Relevant Packages and Datasets

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import collections
from IPython import display
from matplotlib import pyplot as plt
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
tf.logging.set_verbosity(tf.logging.ERROR)

# Add some convenience functions to Pandas DataFrame.
pd.options.display.max_rows = 10
pd.options.display.float_format = '{:.3f}'.format
```

```
WARNING:tensorflow:From C:\Users\laram\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-
packages\tensorflow\python\compat\v2_compat.py:111: disable_resource_variables (from
tensorflow.python.ops.variable_scope) is deprecated and will be removed in a future version.
Instructions for updating:
non-resource variables are not supported in the long term
```

```
user_artists = pd.read_csv('../data/user_artists_ratings.csv')
user_friends = pd.read_csv('../data/user_friends.csv')
user_taggedartists = pd.read_csv('../data/user_taggedartists.csv')
artists = pd.read_csv('../data/artists.csv')
tags = pd.read_csv('../data/tags.csv')
artists_tags = pd.read_csv('../data/artists_tags.csv')
```

Creating Functions to Build Sparse Tensor and Calculate MSE

```
def build_rating_sparse_tensor(user_artists_df):

    indices = user_artists_df[['userID', 'artistID']].values
    values = user_artists_df['rating'].values

    return tf.SparseTensor(
        indices=indices,
        values=values,
        dense_shape=[len(user_artists.userID.unique()), artists.shape[0]])
```

```
def sparse_mean_square_error(sparse_ratings, user_embeddings, music_embeddings):

    predictions = tf.reduce_sum(
        tf.gather(user_embeddings, sparse_ratings.indices[:, 0]) *
        tf.gather(music_embeddings, sparse_ratings.indices[:, 1]),
        axis=1)
    loss = tf.losses.mean_squared_error(sparse_ratings.values, predictions)
    return loss
```

Creating Functions to Build a Collaborative-Filtering Model

```python
class CFModel(object):
    """Simple class that represents a collaborative filtering model"""
    def __init__(self, embedding_vars, loss, metrics=None):
        """Initializes a CFModel.
        Args:
          embedding_vars: A dictionary of tf.Variables.
          loss: A float Tensor. The loss to optimize.
          metrics: optional list of dictionaries of Tensors. The metrics in each
            dictionary will be plotted in a separate figure during training.
        """
        self._embedding_vars = embedding_vars
        self._loss = loss
        self._metrics = metrics
        self._embeddings = {k: None for k in embedding_vars}
        self._session = None

    @property
    def embeddings(self):
        """The embeddings dictionary."""
        return self._embeddings

    def train(self, num_iterations=100, learning_rate=1.0, plot_results=True,
              optimizer=tf.train.GradientDescentOptimizer):
        """Trains the model.
        Args:
          iterations: number of iterations to run.
          learning_rate: optimizer learning rate.
          plot_results: whether to plot the results at the end of training.
          optimizer: the optimizer to use. Default to GradientDescentOptimizer.
        Returns:
          The metrics dictionary evaluated at the last iteration.
        """
        with self._loss.graph.as_default():
            opt = optimizer(learning_rate)
            train_op = opt.minimize(self._loss)
            local_init_op = tf.group(
                tf.variables_initializer(opt.variables()),
                tf.local_variables_initializer())
            if self._session is None:
                self._session = tf.Session()
                with self._session.as_default():
                    self._session.run(tf.global_variables_initializer())
                    self._session.run(tf.tables_initializer())
                    tf.train.start_queue_runners()

        with self._session.as_default():
            local_init_op.run()
            iterations = []
            metrics = self._metrics or ({},)
            metrics_vals = [collections.defaultdict(list) for _ in self._metrics]

            # Train and append results.
            for i in range(num_iterations + 1):
                _, results = self._session.run((train_op, metrics))
                if (i % 10 == 0) or i == num_iterations:
                    print("\r iteration %d: " % i + ", ".join(
                        ["%s=%f" % (k, v) for r in results for k, v in r.items()]),
                          end='')
                    iterations.append(i)
                    for metric_val, result in zip(metrics_vals, results):
                        for k, v in result.items():
                            metric_val[k].append(v)

            for k, v in self._embedding_vars.items():
                self._embeddings[k] = v.eval()

            if plot_results:
                # Plot the metrics.
                num_subplots = len(metrics)+1
                fig = plt.figure()
                fig.set_size_inches(num_subplots*10, 8)
                for i, metric_vals in enumerate(metrics_vals):
                    ax = fig.add_subplot(1, num_subplots, i+1)
                    for k, v in metric_vals.items():
                        ax.plot(iterations, v, label=k)
                    ax.set_xlim([1, num_iterations])
                    ax.set_xlabel("Number of Iterations")
                    ax.set_ylabel("Mean Squared Error")
                    ax.set_title("Training and Testing Errors")
                    ax.legend()
            return results
```

```
def build_model(ratings, embedding_dim=3, init_stddev=1.):

    # Split the ratings DataFrame into train and test
    train_ratings, test_ratings = split_dataframe(ratings)
    # SparseTensor representation of the train and test datasets.
    A_train = build_rating_sparse_tensor(train_ratings)
    A_test = build_rating_sparse_tensor(test_ratings)
    # Initialize the embeddings using a normal distribution.
    U = tf.Variable(tf.random_normal(
        [A_train.dense_shape[0], embedding_dim], stddev=init_stddev))
    V = tf.Variable(tf.random_normal(
        [A_train.dense_shape[1], embedding_dim], stddev=init_stddev))
    train_loss = sparse_mean_square_error(A_train, U, V)
    test_loss = sparse_mean_square_error(A_test, U, V)
    metrics = {
        'train_error': train_loss,
        'test_error': test_loss
    }
    embeddings = {
        "userID": U,
        "artistID": V
    }
    return CFModel(embeddings, train_loss, [metrics])
```

```
def split_dataframe(df, holdout_fraction=0.1):

    test = df.sample(frac=holdout_fraction, replace=False)
    train = df[~df.index.isin(test.index)]
    return train, test
```

Building and Training the Model

```
model = build_model(user_artists, embedding_dim=30, init_stddev=0.5)
model.train(num_iterations=2000, learning_rate=10.)
```

```
iteration 0: train_error=7.674745, test_error=7.676869

iteration 10: train_error=6.973046, test_error=7.464542

iteration 20: train_error=6.428425, test_error=7.316956

iteration 30: train_error=5.963654, test_error=7.190217

iteration 40: train_error=5.533082, test_error=7.055270

iteration 50: train_error=5.107881, test_error=6.890791

iteration 60: train_error=4.678211, test_error=6.688717

iteration 70: train_error=4.255985, test_error=6.460946

iteration 80: train_error=3.861839, test_error=6.230646

iteration 90: train_error=3.507273, test_error=6.014506

iteration 100: train_error=3.193339, test_error=5.818508

iteration 110: train_error=2.917263, test_error=5.643669

iteration 120: train_error=2.675326, test_error=5.489468

iteration 130: train_error=2.463389, test_error=5.354567

iteration 140: train_error=2.277289, test_error=5.237117

iteration 150: train_error=2.113181, test_error=5.135093

iteration 160: train_error=1.967715, test_error=5.046528

iteration 170: train_error=1.838059, test_error=4.969631

iteration 180: train_error=1.721860, test_error=4.902824

iteration 190: train_error=1.617180, test_error=4.844743

iteration 200: train_error=1.522428, test_error=4.794218

iteration 210: train_error=1.436290, test_error=4.750252

iteration 220: train_error=1.357679, test_error=4.711993

iteration 230: train_error=1.285689, test_error=4.678712

iteration 240: train_error=1.219560, test_error=4.649786

iteration 250: train_error=1.158646, test_error=4.624678

iteration 260: train_error=1.102397, test_error=4.602925

iteration 270: train_error=1.050340, test_error=4.584130
```

```
iteration 280: train_error=1.002065, test_error=4.567944

iteration 290: train_error=0.957216, test_error=4.554067

iteration 300: train_error=0.915479, test_error=4.542235

iteration 310: train_error=0.876578, test_error=4.532219

iteration 320: train_error=0.840269, test_error=4.523817

iteration 330: train_error=0.806334, test_error=4.516850

iteration 340: train_error=0.774578, test_error=4.511162

iteration 350: train_error=0.744826, test_error=4.506617

iteration 360: train_error=0.716921, test_error=4.503088

iteration 370: train_error=0.690719, test_error=4.500470

iteration 380: train_error=0.666093, test_error=4.498664

iteration 390: train_error=0.642924, test_error=4.497585

iteration 400: train_error=0.621107, test_error=4.497155

iteration 410: train_error=0.600542, test_error=4.497307

iteration 420: train_error=0.581141, test_error=4.497978

iteration 430: train_error=0.562822, test_error=4.499113

iteration 440: train_error=0.545510, test_error=4.500663

iteration 450: train_error=0.529134, test_error=4.502584

iteration 460: train_error=0.513633, test_error=4.504835

iteration 470: train_error=0.498946, test_error=4.507381

iteration 480: train_error=0.485019, test_error=4.510190

iteration 490: train_error=0.471802, test_error=4.513234

iteration 500: train_error=0.459249, test_error=4.516484

iteration 510: train_error=0.447317, test_error=4.519921

iteration 520: train_error=0.435965, test_error=4.523520

iteration 530: train_error=0.425158, test_error=4.527264

iteration 540: train_error=0.414859, test_error=4.531136

iteration 550: train_error=0.405039, test_error=4.535119
```

```
iteration 560: train_error=0.395666, test_error=4.539202
iteration 570: train_error=0.386714, test_error=4.543370
iteration 580: train_error=0.378157, test_error=4.547613
iteration 590: train_error=0.369971, test_error=4.551921
iteration 600: train_error=0.362134, test_error=4.556286
iteration 610: train_error=0.354625, test_error=4.560698
iteration 620: train_error=0.347425, test_error=4.565149
iteration 630: train_error=0.340517, test_error=4.569636
iteration 640: train_error=0.333882, test_error=4.574151
iteration 650: train_error=0.327507, test_error=4.578688
iteration 660: train_error=0.321376, test_error=4.583243
iteration 670: train_error=0.315475, test_error=4.587811
iteration 680: train_error=0.309793, test_error=4.592391
iteration 690: train_error=0.304316, test_error=4.596976
iteration 700: train_error=0.299035, test_error=4.601566
iteration 710: train_error=0.293938, test_error=4.606156
iteration 720: train_error=0.289016, test_error=4.610745
iteration 730: train_error=0.284260, test_error=4.615329
iteration 740: train_error=0.279662, test_error=4.619909
iteration 750: train_error=0.275212, test_error=4.624481
iteration 760: train_error=0.270905, test_error=4.629044
iteration 770: train_error=0.266732, test_error=4.633598
iteration 780: train_error=0.262687, test_error=4.638139
iteration 790: train_error=0.258765, test_error=4.642668
iteration 800: train_error=0.254959, test_error=4.647182
iteration 810: train_error=0.251263, test_error=4.651683
iteration 820: train_error=0.247673, test_error=4.656168
iteration 830: train_error=0.244184, test_error=4.660638
```

```
iteration 840: train_error=0.240792, test_error=4.665092

iteration 850: train_error=0.237491, test_error=4.669528

iteration 860: train_error=0.234278, test_error=4.673947

iteration 870: train_error=0.231150, test_error=4.678349

iteration 880: train_error=0.228101, test_error=4.682732

iteration 890: train_error=0.225131, test_error=4.687097

iteration 900: train_error=0.222234, test_error=4.691444

iteration 910: train_error=0.219407, test_error=4.695772

iteration 920: train_error=0.216649, test_error=4.700081

iteration 930: train_error=0.213957, test_error=4.704371

iteration 940: train_error=0.211327, test_error=4.708642

iteration 950: train_error=0.208757, test_error=4.712893

iteration 960: train_error=0.206246, test_error=4.717125

iteration 970: train_error=0.203790, test_error=4.721338

iteration 980: train_error=0.201389, test_error=4.725532

iteration 990: train_error=0.199039, test_error=4.729705

iteration 1000: train_error=0.196739, test_error=4.733860

iteration 1010: train_error=0.194487, test_error=4.737995

iteration 1020: train_error=0.192282, test_error=4.742110

iteration 1030: train_error=0.190122, test_error=4.746207

iteration 1040: train_error=0.188006, test_error=4.750283

iteration 1050: train_error=0.185931, test_error=4.754341

iteration 1060: train_error=0.183897, test_error=4.758379

iteration 1070: train_error=0.181902, test_error=4.762398

iteration 1080: train_error=0.179945, test_error=4.766398

iteration 1090: train_error=0.178025, test_error=4.770379

iteration 1100: train_error=0.176141, test_error=4.774340

iteration 1110: train_error=0.174291, test_error=4.778284
```

```
iteration 1120: train_error=0.172475, test_error=4.782208
```

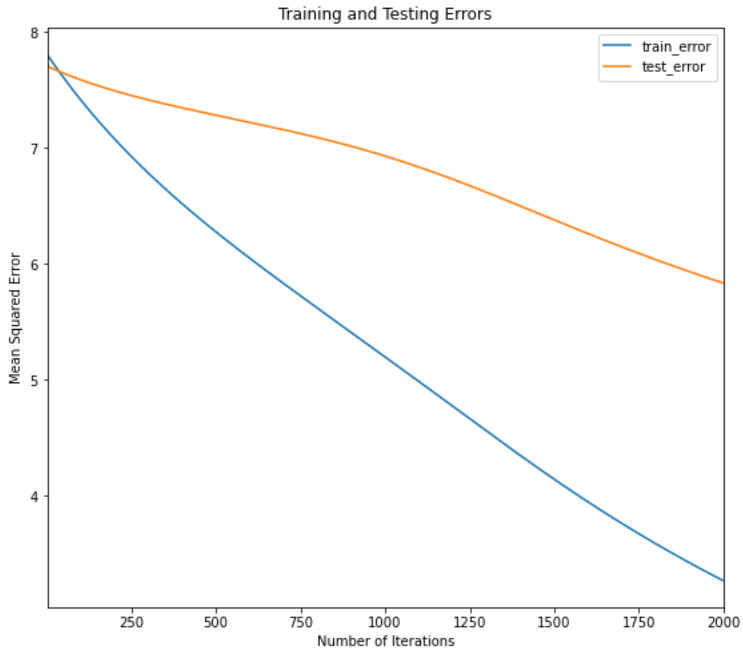
```
iteration 1130: train_error=0.170691, test_error=4.786114
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_15636/460896045.py in <module>
      1 model = build_model(user_artists, embedding_dim=30, init_stddev=0.5)
----> 2 model.train(num_iterations=2000, learning_rate=10.)

~\AppData\Local\Temp/ipykernel_15636/701340561.py in train(self, num_iterations,
learning_rate, plot_results, optimizer)
     52             # Train and append results.
     53             for i in range(num_iterations + 1):
---> 54                 _, results = self._session.run((train_op, metrics))
     55                 if (i % 10 == 0) or i == num_iterations:
     56                     print("\r iteration %d: " % i + ", ".join(

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-
packages\tensorflow\python\client\session.py in run(self, fetches, feed_dict, options,
run_metadata)
    968
    969     try:
--> 970       result = self._run(None, fetches, feed_dict, options_ptr,
    971                          run_metadata_ptr)
    972       if run_metadata:

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-
packages\tensorflow\python\client\session.py in _run(self, handle, fetches, feed_dict,
options, run_metadata)
   1191     # or if the call is a partial run that specifies feeds.
   1192     if final_fetches or final_targets or (handle and feed_dict_tensor):
-> 1193       results = self._do_run(handle, final_targets, final_fetches,
   1194                              feed_dict_tensor, options, run_metadata)
   1195     else:

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-
packages\tensorflow\python\client\session.py in _do_run(self, handle, target_list, fetch_list,
feed_dict, options, run_metadata)
   1371
   1372     if handle is None:
-> 1373       return self._do_call(_run_fn, feeds, fetches, targets, options,
   1374                            run_metadata)
   1375     else:

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-
packages\tensorflow\python\client\session.py in _do_call(self, fn, *args)
   1378   def _do_call(self, fn, *args):
   1379     try:
-> 1380       return fn(*args)
   1381     except errors.OpError as e:
   1382       message = compat.as_text(e.message)

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-
packages\tensorflow\python\client\session.py in _run_fn(feed_dict, fetch_list, target_list,
options, run_metadata)
   1361       # Ensure any changes to the graph are reflected in the runtime.
   1362       self._extend_graph()
-> 1363       return self._call_tf_sessionrun(options, feed_dict, fetch_list,
   1364                                       target_list, run_metadata)
   1365

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-
packages\tensorflow\python\client\session.py in _call_tf_sessionrun(self, options, feed_dict,
fetch_list, target_list, run_metadata)
   1454   def _call_tf_sessionrun(self, options, feed_dict, fetch_list, target_list,
   1455                           run_metadata):
-> 1456     return tf_session.TF_SessionRun_wrapper(self._session, options, feed_dict,
   1457                                             fetch_list, target_list,
   1458                                             run_metadata)

KeyboardInterrupt:
```

```
model = build_model(user_artists, embedding_dim=30, init_stddev=0.5)
model.train(num_iterations=2000, learning_rate=.5)
```

```
iteration 2000: train_error=3.266350, test_error=5.835364
```

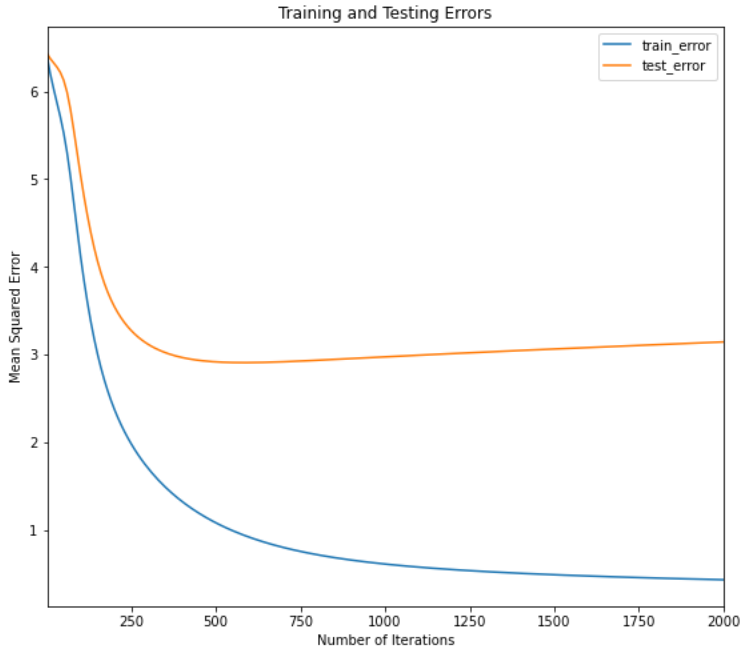
```
[{'train_error': 3.26635, 'test_error': 5.8353643}]
```




```
model = build_model(user_artists, embedding_dim=10, init_stddev=0.5)
model.train(num_iterations=2000, learning_rate=10)
```

```
iteration 2000: train_error=0.434694, test_error=3.144789
```

```
[{'train_error': 0.43469417, 'test_error': 3.1447892}]
```



Inspect Embeddings

```
DOT = 'dot'
COSINE = 'cosine'
def compute_scores(query_embedding, item_embeddings, measure=DOT):

    u = query_embedding
    V = item_embeddings
    if measure == COSINE:
        V = V / np.linalg.norm(V, axis=1, keepdims=True)
        u = u / np.linalg.norm(u)
    scores = u.dot(V.T)
    return scores
```

```
def artist_neighbors(model, title_substring, measure=DOT, k=6):
    ids =  artists[artists['name'].str.contains(title_substring)].index.values
    titles = artists.iloc[ids]['name'].values
    if len(titles) == 0:
        raise ValueError("Found no artist with title %s" % title_substring)
    print("Nearest neighbors of : %s." % titles[0])
    if len(titles) > 1:
        print("[Found more than one matching artist. Other candidates: {}]".format(
            ", ".join(titles[1:])))
    artistID = ids[0]
    scores = compute_scores(
        model.embeddings["artistID"][artistID], model.embeddings["artistID"],
        measure)
    score_key = measure + ' score'
    df = pd.DataFrame({
        score_key: list(scores),
        'names': artists['name'],
    })
    display.display(df.sort_values([score_key], ascending=False).head(k))
```

```
artist_neighbors(model, "Kanye", DOT)
```

```
Nearest neighbors of : Kanye West.
[Found more than one matching artist. Other candidates: Kanye West & Jay-Z, Kanye West,
Beyoncé & Charlie Wilson, Kanye West feat. Pusha T MpTri.Net, Eminem feat. Drake, Kanye West,
Lil Wayne, The-Dream f. Kanye West, Jamie Foxx feat. Kanye West and The-Dream]
```

	dot score	names
3833	3.895	Bad Boys Blue
3227	3.765	Marillion
257	3.647	Jill Scott
8177	3.606	Axel Fernando
1	3.570	Diary of Dreams
61	3.489	Madonna

```
artist_neighbors(model, "Kanye", COSINE)
```

```
Nearest neighbors of : Kanye West.
[Found more than one matching artist. Other candidates: Kanye West & Jay-Z, Kanye West,
Beyoncé & Charlie Wilson, Kanye West feat. Pusha T MpTri.Net, Eminem feat. Drake, Kanye West,
Lil Wayne, The-Dream f. Kanye West, Jamie Foxx feat. Kanye West and The-Dream]
```

	cosine score	names
325	1.000	Kanye West
9110	0.947	Airbag
176	0.931	Keane
2570	0.924	Auto!Automatic!!
683	0.923	Florence + the Machine
5825	0.922	Cash Cash

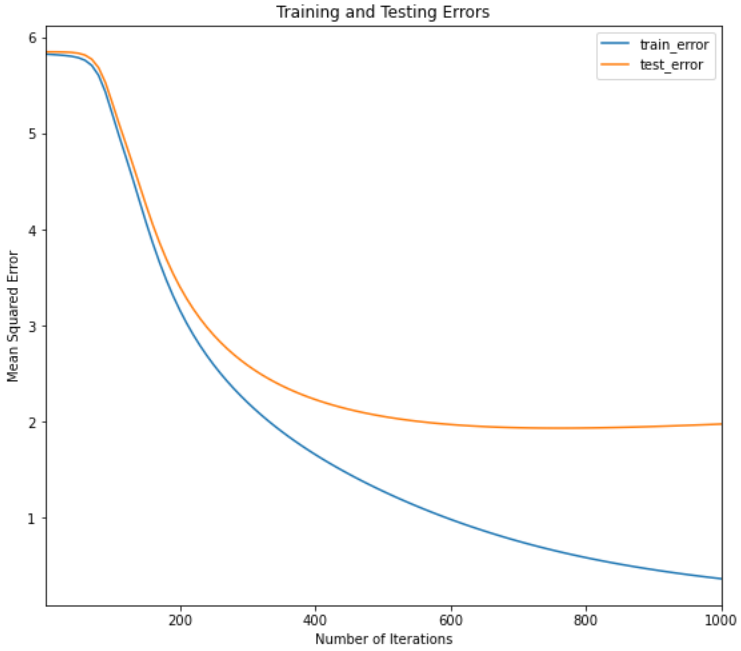
```
model_lowinit = build_model(user_artists, embedding_dim=30, init_stddev=0.05)
model_lowinit.train(num_iterations=1000, learning_rate=10.)
artist_neighbors(model_lowinit, "Kanye", DOT)
artist_neighbors(model_lowinit, "Kanye", COSINE)
```

```
 iteration 1000: train_error=0.365735, test_error=1.977160Nearest neighbors of : Kanye West.
[Found more than one matching artist. Other candidates: Kanye West & Jay-Z, Kanye West,
Beyoncé & Charlie Wilson, Kanye West feat. Pusha T MpTri.Net, Eminem feat. Drake, Kanye West,
Lil Wayne, The-Dream f. Kanye West, Jamie Foxx feat. Kanye West and The-Dream]
```

	dot score	names
325	6.341	Kanye West
300	5.148	Black Eyed Peas
894	4.058	Amy Winehouse
458	3.739	3OH!3
50	3.653	Daft Punk
455	3.374	Miley Cyrus

```
Nearest neighbors of : Kanye West.
[Found more than one matching artist. Other candidates: Kanye West & Jay-Z, Kanye West,
Beyoncé & Charlie Wilson, Kanye West feat. Pusha T MpTri.Net, Eminem feat. Drake, Kanye West,
Lil Wayne, The-Dream f. Kanye West, Jamie Foxx feat. Kanye West and The-Dream]
```

	cosine score	names
325	1.000	Kanye West
11036	0.724	Eminem feat. Drake, Kanye West, Lil Wayne
6179	0.683	Lady GaGa Ft. Beyoncé
9154	0.679	Nancy Ajram
277	0.676	Ana Carolina
4606	0.658	John Legend

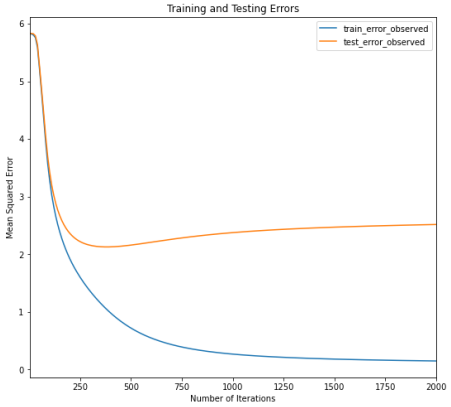


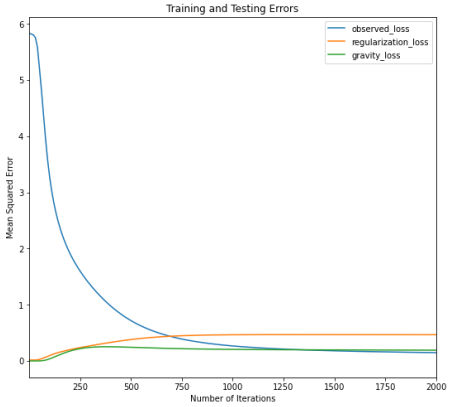
```
def gravity(U, V):
    """Creates a gravity loss given two embedding matrices."""
    return 1. / (U.shape[0].value*V.shape[0].value) * tf.reduce_sum(
        tf.matmul(U, U, transpose_a=True) * tf.matmul(V, V, transpose_a=True))

def build_regularized_model(
    ratings,
    embedding_dim=3,
    regularization_coeff=.1,
    gravity_coeff=1.,
    init_stddev=0.1):
    # Split the ratings DataFrame into train and test.
    train_ratings, test_ratings = split_dataframe(ratings)
    # SparseTensor representation of the train and test datasets.
    A_train = build_rating_sparse_tensor(train_ratings)
    A_test = build_rating_sparse_tensor(test_ratings)
    U = tf.Variable(tf.random_normal(
        [A_train.dense_shape[0], embedding_dim], stddev=init_stddev))
    V = tf.Variable(tf.random_normal(
        [A_train.dense_shape[1], embedding_dim], stddev=init_stddev))

    error_train = sparse_mean_square_error(A_train, U, V)
    error_test = sparse_mean_square_error(A_test, U, V)
    gravity_loss = gravity_coeff * gravity(U, V)
    regularization_loss = regularization_coeff * (
        tf.reduce_sum(U*U)/U.shape[0].value + tf.reduce_sum(V*V)/V.shape[0].value)
    total_loss = error_train + regularization_loss + gravity_loss
    losses = {
        'train_error_observed': error_train,
        'test_error_observed': error_test,
    }
    loss_components = {
        'observed_loss': error_train,
        'regularization_loss': regularization_loss,
        'gravity_loss': gravity_loss,
    }
    embeddings = {"userId": U, "artistID": V}

    return CFModel(embeddings, total_loss, [losses, loss_components]), U, V
```

```
reg_model, u, v = build_regularized_model(
    user_artists, regularization_coeff=0.1, gravity_coeff=1.0, embedding_dim=35,
    init_stddev=.05)
reg_model.train(num_iterations=2000, learning_rate=20.)
```

```
 iteration 2000: train_error_observed=0.148672, test_error_observed=2.516972,
observed_loss=0.148672, regularization_loss=0.467933, gravity_loss=0.192609
```

```
[{'train_error_observed': 0.14867209, 'test_error_observed': 2.5169723},
 {'observed_loss': 0.14867209,
  'regularization_loss': 0.46793318,
  'gravity_loss': 0.19260909}]
```






```
artist_neighbors(reg_model, "Kanye", DOT)
artist_neighbors(reg_model, "Kanye", COSINE)
```

```
Nearest neighbors of : Kanye West.
[Found more than one matching artist. Other candidates: Kanye West & Jay-Z, Kanye West,
Beyoncé & Charlie Wilson, Kanye West feat. Pusha T MpTri.Net, Eminem feat. Drake, Kanye West,
Lil Wayne, The-Dream f. Kanye West, Jamie Foxx feat. Kanye West and The-Dream]
```

	dot score	names
325	21.095	Kanye West
221	15.184	The Beatles
283	14.188	Britney Spears
294	12.577	Katy Perry
83	11.557	Lady Gaga
228	11.408	Nirvana

```
Nearest neighbors of : Kanye West.
[Found more than one matching artist. Other candidates: Kanye West & Jay-Z, Kanye West,
Beyoncé & Charlie Wilson, Kanye West feat. Pusha T MpTri.Net, Eminem feat. Drake, Kanye West,
Lil Wayne, The-Dream f. Kanye West, Jamie Foxx feat. Kanye West and The-Dream]
```

	cosine score	names
325	1.000	Kanye West
1072	0.687	Ace of Base
1600	0.666	Lykke Li
965	0.659	Amy Studt
11027	0.650	drake and lil wayne
3425	0.650	[unknown]

Using the Recommender Systems

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from basic_recommender_system import *
from content_based_recommender import *
```

These recommender systems can be used if you create a local version of this repo and change some of the variables in this noetbook.

Guideline to use the Basic Recommender System

The **basic recommender system** can be called using the function Find_The_Most_Popular_Music(). The parameters are as follows:

1. method : This is a string value, and can be either:
    - most_users (default): Returns the two artists with the highest number of users
    - most_listens : Takes the top 10% of artists with the highest user count and returns the two with the highest number of listens
    - star_rating : Takes the top 10% of artists with the highest listens and removes artists with less than 5 users and returns the two artists with the highest average rating
    - custom : This is a customised option for recommendations. You get to decide how to filter the artists (top 10% of artists with the highest user count, top 25% of users with the highest listen count, top 5% of users with the highest star-ratings, etc.). It also allows you to choose the final basis for the recommendation (user_count, listen_count or star_rating). It allows for the following optional arguments:
2. (optional) percentage_of_artists : This defaults to 10%. It takes the top 10% of artists based on your choice of filtering.
3. (optional) percentage_basis : This can be any of the following:
    - user_count (default) : This takes the top percentage of artists based on the number of users listening to them
    - listen_count : This takes the top percentage of artists based on the number of times they've been listened to
    - star_rating : This takes the top percentage of artists based on their average star-rating
4. (optional) recommendation_basis : This can be any of the following:

- user_count : This returns the top two artists based on the number of users listening to them
- listen_count : This takes the top two artists based on the number of times they've been listened to
- star_rating (default) : This takes the top two artists based on their average star-rating

```
Find_The_Most_Popular_Music("most_listens")
```

```
We think you might like Britney Spears, and you can view their profile through this link:
http://www.last.fm/music/Britney+Spears
You might also like Depeche Mode, and you can view their profile through this link:
http://www.last.fm/music/Depeche+Mode
```

```
Find_The_Most_Popular_Music("most_users")
```

```
We think you might like Lady Gaga, and you can view their profile through this link:
http://www.last.fm/music/Lady+Gaga
You might also like Britney Spears, and you can view their profile through this link:
http://www.last.fm/music/Britney+Spears
```

```
Find_The_Most_Popular_Music("star_rating")
```

```
We think you might like Duran Duran, and you can view their profile through this link:
http://www.last.fm/music/Duran+Duran
You might also like The Smashing Pumpkins, and you can view their profile through this link:
http://www.last.fm/music/The+Smashing+Pumpkins
```

```
Find_The_Most_Popular_Music(
    method="custom",
    percentage_of_artists=10,
    percentage_basis="star_rating",
    recommendation_basis="most_users")
```

```
We think you might like Britney Spears, and you can view their profile through this link:
http://www.last.fm/music/Britney+Spears
You might also like Depeche Mode, and you can view their profile through this link:
http://www.last.fm/music/Depeche+Mode
```

This custom recommendation query retrieved the top 10% of artists that received the highest average user rating, and returned Britney Spears and Depeche Mode as the artists who have the most users listening.

Guideline to use the Content-Based Recommender System

There are two recommender systems created:

1. A recommender system that calculates similarity based off only the top tag assigned to each artist.
    - This can be called using content_based_recommendation_1(artists_name, artists_tags, tags)
2. A recommender system that calculates similarity based off their name and the top five tags assigned to each artist.
    - This can be called using content_based_recommendation_2(artists_name, artists_tags, tags)

Note: These functions may not run depending on system resources (Memory Error). If this is the case, you can check the notebook 'Content-Based Recommender System Workings' as there are examples shown there.

```
artists_tags = pd.read_csv('../data/artists_tags.csv')
tags = pd.read_csv('../data/tags.csv')
```

```
content_based_recommendation_1('Kanye West', artists_tags, tags)
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_32860/3187041587.py in <module>
----> 1 content_based_recommendation_1('Kanye West', artists_tags, tags)

~\CA4015\CA4015-Recommender-System\jupyter-book\content_based_recommender.py in
content_based_recommendation_1(artist_name, artist_tags_df, tags_df)
     56
     57     # Defining the cosine similarity and storing in a matrix
---> 58     cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
     59
     60     # Creating an inverted index for easier calling of artist names

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\sklearn\metrics\pairwise.py
in linear_kernel(X, Y, dense_output)
   1003     """
   1004     X, Y = check_pairwise_arrays(X, Y)
-> 1005     return safe_sparse_dot(X, Y.T, dense_output=dense_output)
   1006
   1007

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\sklearn\utils\validation.py
in inner_f(*args, **kwargs)
     61             extra_args = len(args) - len(all_args)
     62             if extra_args <= 0:
---> 63                 return f(*args, **kwargs)
     64
     65             # extra_args > 0

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\sklearn\utils\extmath.py in
safe_sparse_dot(a, b, dense_output)
    154     if (sparse.issparse(a) and sparse.issparse(b)
    155             and dense_output and hasattr(ret, "toarray")):
--> 156         return ret.toarray()
    157     return ret
    158

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\scipy\sparse\compressed.py
in toarray(self, order, out)
   1029         if out is None and order is None:
   1030             order = self._swap('cf')[0]
-> 1031         out = self._process_toarray_args(order, out)
   1032         if not (out.flags.c_contiguous or out.flags.f_contiguous):
   1033             raise ValueError('Output array must be C or F contiguous')

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\scipy\sparse\base.py in
_process_toarray_args(self, order, out)
   1200             return out
   1201         else:
-> 1202             return np.zeros(self.shape, dtype=self.dtype, order=order)
   1203
   1204

MemoryError: Unable to allocate 2.32 GiB for an array with shape (17632, 17632) and data type
float64
```

```
content_based_recommendation_2('Kanye West', artists_tags, tags)
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_42284/3896620694.py in <module>
----> 1 content_based_recommendation_2('Kanye West', artists_tags, tags)

~\CA4015\CA4015-Recommender-System\jupyter-book\content_based_recommender.py in
content_based_recommendation_2(artist_name, artist_tags_df, tags_df)
     82
     83     # Defining the cosine similarity and storing in a matrix
---> 84     cosine_sim = cosine_similarity(count_matrix, count_matrix)
     85
     86     # Creating an inverted index for easier calling of artist names

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\sklearn\metrics\pairwise.py
in cosine_similarity(X, Y, dense_output)
   1186         Y_normalized = normalize(Y, copy=True)
   1187
-> 1188     K = safe_sparse_dot(X_normalized, Y_normalized.T,
   1189                         dense_output=dense_output)
   1190

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\sklearn\utils\validation.py
in inner_f(*args, **kwargs)
     61             extra_args = len(args) - len(all_args)
     62             if extra_args <= 0:
---> 63                 return f(*args, **kwargs)
     64
     65             # extra_args > 0

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\sklearn\utils\extmath.py in
safe_sparse_dot(a, b, dense_output)
    154     if (sparse.issparse(a) and sparse.issparse(b)
    155             and dense_output and hasattr(ret, "toarray")):
--> 156         return ret.toarray()
    157     return ret
    158

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\scipy\sparse\compressed.py
in toarray(self, order, out)
   1029         if out is None and order is None:
   1030             order = self._swap('cf')[0]
-> 1031         out = self._process_toarray_args(order, out)
   1032         if not (out.flags.c_contiguous or out.flags.f_contiguous):
   1033             raise ValueError('Output array must be C or F contiguous')

~\AppData\Local\Continuum\anaconda3\envs\ca4015\lib\site-packages\scipy\sparse\base.py in
_process_toarray_args(self, order, out)
   1200             return out
   1201         else:
-> 1202             return np.zeros(self.shape, dtype=self.dtype, order=order)
   1203
   1204

MemoryError: Unable to allocate 2.32 GiB for an array with shape (17632, 17632) and data type
float64
```

By Lara Murphy